Verifikation Nebenläufiger Programme: Verfeinerung, Synchronisation, Sequenziellisierung[1]

Bernhard Kragl[2]

Abstract: Unsere Gesellschaft vertraut maßgeblich und stetig zunehmend auf verteilte IT Systeme. Allerdings sind Entwurf und Verifikation nebenläufiger Programme berüchtigt komplizierte, zeitintensive und fehleranfällige Aufgaben, selbst für Fachexperten. Der Grund dafür ist die enorme (unendliche) Menge an verzahnten Ausführungen nebenläufiger Programme. All diese Ausführungen müssen in einem formalen Korrektheitsbeweis erfasst und berücksichtigt werden. Solche Beweise basieren bekanntermaßen auf induktiven Invarianten über den globalen Programmzustand. Die Erarbeitung induktiver Invarianten für konkrete Implementierungen ist zwar theoretisch möglich, aber praktisch undenkbar. In dieser Dissertation präsentieren wir eine Verifikationsmethodik basierend auf dem Konzept der schrittweisen Verfeinerung, welche die Konstruktion formaler Korrektheitsbeweise grundlegend vereinfacht. Wir präsentieren eine Programmiersprache zur kompakten Beschreibung von Beweisen über mehrere Abstraktionsebenen. Eine mächtige Beweisregel unterteilt das Verifikationsproblem in zahlreiche automatisierbare Unterprobleme. Neue Beweistaktiken für asynchrone Programme ermöglichen die Reduktion von komplexen nebenläufigen Ausführungen zu simplen sequenziellen Ausführungen. Unsere Methodik ist in unserem statischen Analyseprogramm CIVL implementiert und dessen Effektivität wird in zahlreichen Fallstudien demonstriert. CIVL wurde bereits in mehreren Publikationen anderer Forscher verwendet.

1 Einführung

Nebenläufigkeit ist in heutigen Computersystemen allgegenwärtig. Geografisch verteilte Datenzentren benötigen fehlertolerante verteilte Algorithmen zur Sicherstellung eines konsistenten Systemzustandes; massiv-parallele Supercomputer ermöglichen gigantische wissenschaftliche Berechnungen; Mobil- und Webanwendungen verwenden ereignisgetriebene asynchrone Programmierung um ein schnelles und flüssiges Benutzererlebnis zu bieten; die Kontrollsoftware von eingebetteten Systemen muss auf asynchrone Ereignisse der realen Welt reagieren; nur um einige Beispiele zu nennen. In nebenläufigen Programm erfolgen mehrere Berechnungsschritte gleichzeitig. Zum Beispiel können auf einem Mehrkernprozessor mehrere Threads (je einer pro Prozessorkern) gleichzeitig laufen, welche Daten über ein geteiltes Speichersystem austauschen. In einem verteilten System kommunizieren Prozesse auf unabhängigen Computern durch den austausch von Nachrichten welche über ein Netzwerk geschickt werden. Formell verstehen wir "gleichzeitig" als eine Verzahnung der atomaren (d.h. nicht weiter in Teilschritte unterteilbare) Aktionen der nebenläufigen Berechnungen. Sind zum Beispiel $A_1;A_2;A_3$ die atomaren Aktionen von Prozess A und $B_1;B_2;B_3$ die atomaren Aktionen von Prozess B, so ist $A_1;B_1;B_2;A_2;B_3;A_3$ eine der 20 möglichen verzahnten Ausführungen der beiden Prozesse.

[1] Englischer Titel der Dissertation: "Verifying Concurrent Programs: Refinement, Synchronization, Sequentialization

[2] IST Austria, bernhard.kragl@gmail.com

Die Anzahl an verzahnten Ausführungen steigt exponentiell mit der Anzahl nebenläufiger Prozesse und der Anzahl atomarer Aktionen pro Prozess. Diese explodierende Anzahl an Verzahnungen macht das Nachvollziehen aller möglichen Ausführungen nebenläufiger Programme eine beinahe unmögliche mentale Aufgabe, insbesondere wenn Systeme adaptiert und weiterentwickelt werden. Unvorhergesehene Ausnahmesituationen definitiv auszuschließen ist extrem schwierig. Das Paxos Protokoll [La98], welches im Kern beinahe jedes replizierten Systems verwendet wird, ist berüchtigt für seine Komplexität; schon auf Papier [La02a, OO14, vRA15] und erst recht in konkreten Implementierungen [CGR07]. Im Chord Algorithmus [St01] für verteilte Hashtabellen fanden sich über 10 Jahre in allen publizierten Varianten Korrektheitsfehler [Za12].

Bevor wir sagen können *ob* ein System korrekt ist, müssen wir definieren *was es bedeutet* für das System korrekt zu sein; wir brauchen eine *Spezifikation*. Spezifikationen variieren in Form und Gestalt. Zum Beispiel können wir uns interessieren für flache generische Eigenschaften (wie Speichersicherheit oder Deadlockfreiheit), tiefe funktionale Eigenschaften, temporale Eigenschaften, Hypereigenschaften (wie Datenschutz), etc. In dieser Dissertation zielen wir auf tiefe funktionale Safety-Eigenschaften [MP90] ab, welche in mathematischer Logik ausgedrückt werden. Beispielsweise drückt in einem System mit einer endlichen Menge an Prozessen P die Formel

$$\forall\, p_1, p_2 \in P.\ p_1.\text{hasDecided} \wedge p_2.\text{hasDecided} \implies p_1.\text{decidedValue} = p_2.\text{decidedValue}$$

aus, dass sich die Prozesse auf einen konsistenten Wert einigen müssen (ein Teil des berühmten Konsensusproblems). Wenn sich je zwei beliebige Prozesse p_1 und p_2 für einen Wert entschieden haben, so muss dieser Wert der gleiche sein.

Es gibt viele Ansätze um Vertrauen in die Korrektheit eines Computerprogramms zu bekommen. Etwa durch Testen, statische Analyse, Modellprüfung, oder deduktive Verifikation. Diese Ansätze spannen ein breites Spektrum zwischen Kosten (Zeit, Ressourcen, Expertise, etc.) und Nutzen (stärke der resultierenden Garantie). Der Fokus dieser Dissertation ist das Beweisen starker benutzerdefinierter Spezifikationen *aller* (unendlich vieler) nebenläufiger Ausführungen durch deduktive Verifikation. Dabei ist unser Ziel, einerseits den nötigen manuellen (kreativen) Aufwand so einfach und angenehm als möglich zu gestalten, und andererseits alle automatisierbaren Schritte an einen Computer zu übergeben.

2 Deduktive Verifikation nebenläufiger Programme

Wir wiederholen die fundamentalen Konzepte der *induktiven Invarianten*, der *Reduktion*, und der *Verfeinerung*, und beschreiben unsere Innovationen auf Basis dieser Konzepte.

Induktive Invarianten. Angenommen ein Programm ist modelliert als Transitionssystem $(\mathit{Var}, \mathit{Init}, \mathit{Next}, \mathit{Safe})$, wobei Var die Menge an Programmvariablen, Init das Initialzustandsprädikat über Var, Next das Transitionsprädikat über $\mathit{Var} \cup \mathit{Var}'$, und Safe ein Sicherheitsprädikat über Var ist. Wir wollen sicherstellen, dass keine Programmausführung einen unsicheren Zustand erreichen kann, d.h., für alle Zustandssequenzen $s_1, \ldots, s_n$ mit $s_1 \models \mathit{Init}$

und $s_i, s'_{i+1} \models \mathit{Next}$ für alle Paare von aufeinander folgenden Zuständen soll $s_n \models \mathit{Safe}$ gelten. Wir erreichen dies durch Finden einer *induktiven Invariante*—einem Prädikat *Inv* über *Var* so dass (1) $\mathit{Init} \implies \mathit{Inv}$, (2) $\mathit{Inv} \wedge \mathit{Next} \implies \mathit{Inv}'$, und (3) $\mathit{Inv} \implies \mathit{Safe}$ gelten. Die Invariante gilt im Initialzustand, bleibt durch Zustandsübergänge erhalten, und gilt nicht in unsicheren Zuständen. Daher überapproximiert *Inv* die tatsächlich erreichbaren Zustände und trennt diese von den unsicheren Zuständen.

Die modellierung von Programmen als Transitionssysteme ist adäquat für theoretische Betrachtungen, birgt allerdings Probleme für die Verifikation in der Praxis. Das *Next* Prädikat muss den Kontrollfluss des Programms kodieren, was zu einer Fallunterscheidung über alle möglichen Schritte in allen möglichen Zuständen führt. Diese Fallunterscheidung verstärkt sich dann in *Inv*, was das Auffinden und Formulieren von induktiven Invarianten zu einem äußerst komplizierten und mühsamen Unterfangen macht.

Für sequentielle Programme hat Floyd [Fl67] gezeigt wie Programmbeweise durch Annotation des Programmtextes mit *induktiven Zusicherungen* konstruiert werden können. Das resultierende Beweissystem wird heute als *Floyd-Hoare Logik* [Ho69] bezeichnet, und bildet die Basis moderner Programmbeweiser [Ba05]. Aussagen in Floyd-Hoare Logik werden üblicherweise als $\{\varphi\}\, c\, \{\psi\}$ geschrieben, mit der Bedeutung, dass wenn Kommando c in einem Zustand der die *Vorbedingung* φ erfüllt ausgeführt wird und terminiert, dann erfüllt der Endzustand nach der Ausführung von c die *Nachbedingung* ψ.

Owicki und Gries [OG76] erfanden folgende Beweisregel für nebenläufige Programme.

$$\frac{\Psi_1 : \{\varphi_1\}\, c_1\, \{\psi_1\} \quad \Psi_2 : \{\varphi_2\}\, c_2\, \{\psi_2\} \quad \Psi_1, \Psi_2 \text{ interferenzfrei}}{\{\varphi_1 \wedge \varphi_2\}\, c_1 \parallel c_2\, \{\psi_1 \wedge \psi_2\}}$$

Um eine Eigenschaft der parallelen Ausführung von c_1 and c_2 zu beweisen, können wir demnach über beide Kommandos unabhängig schließen. Allerdings müssen die resultierenden Beweise Ψ_1 und Ψ_2 *interferenzfrei* sein, was bedeutet dass keine Aussage in Ψ_1 duch einen Schritt in c_2 invalidiert werden kann, und gleichermaßen für Ψ_2 und c_1. Abbildung 1 zeigt einen Beispielbeweis für $\{x = 0\}\, x := x+1 \parallel x := x+2\, \{x = 3\}$, wo zwei Threads eine gemeinsame Variable x um 1 bzw. 2 erhöhen. Beide Zuweisungen an x werden als atomar angenommen. Im linken Thread wird die Vorbedingung $x = 0$ zu $x = 0 \vee x = 2$ geschwächt, wodurch wir nach $x := x+1$ die Nachbedingung $x = 1 \vee x = 3$ erhalten. Im rechten Thread wird $x = 0$ zu $x = 0 \vee x = 1$ geschwächt, wodurch wir nach $x := x+2$ die Nachbedingung $x = 2 \vee x = 3$ erhalten. Zusammen folgt aus ψ_1 und ψ_2, dass $x = 3$. Die *Interferenzbedingungen* sind rechts in Abbildung 1 gelistet. Zum Beispiel folgt aus $\varphi_1 \wedge \varphi_2$ dass $x = 0$, und daher gilt nach $x := x+2$ dass $x = 2$, was wiederum φ_1 erfüllt.

Es mag überraschend wirken, dass wir trotz Abschwächung von φ_1 und φ_2 die präzise Nachbedingung $x = 3$ aus ψ_1 und ψ_2 wiedergewinnen konnten. Können wir den Beweis aus Abbildung 1 auf folgendes Programm adaptieren, in dem x von beiden Threads um 1 erhöht wird: $\{x = 0\}\, x := x+1 \parallel x := x+1\, \{x = 2\}$. Die Antwort ist nein. Jede Annotation die nur x erwähnt ist entweder zu schwach um die Nachbedingung $x = 2$ zu folgern, oder zu stark um interferenzfrei zu sein. Als Lösung dieses Problems können sogenannte *Hilfsvariablen* für Beweiszwecke in das Programm eingeführt werden [Ow75]. Allerdings müssen diese Hilfsvariablen zumeist die Kontrollpunkte aller Threads kodieren, was

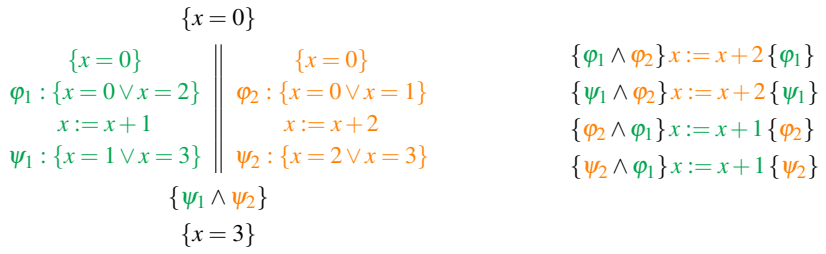

Abbildung 1: Owicki-Gries Beweis eines simplen nebenläufigen Programms.

wiederum (wie bei flachen Transitionssystemen) zu exzessiven Fallunterscheidungen in den Beweisannotationen führt. Dies ist ein massives praktisches Problem für Beweise von komplexen realistischen Programmen.

In dieser Dissertation behandeln wir das Hauptproblem der Verifikation von nebenläufigen Programmen—der Konstruktion von induktiven Invarianten—durch Bereitstellung neuer Techniken und Methodologien zur Unterstützung der interaktiven Beweiskonstruktion. Der Fokus liegt auf Möglichkeiten zur Dekomposition und Strukturierung von Beweisen, wodurch die mentale Aufgabe in überschaubare Teilprobleme zerlegt wird, und der Beweisprozess angenehmer und produktiver gestaltet wird.

Reduktion. Nebenläufige Berechnungen sind normalerweise nicht völlig unabhängig. Trotz vieler möglicher Verzahnungen ist zu erwarten, dass viele dieser Verzahnungen "äquivalent" sind, also zum gleichen Resultat führen. Zwei typische Anwendungen von Nebenläufigkeit sind (1) die Beschleunigung von Berechnungen, und (2) die Zustandsreplikation um Fehler zu tolerieren. In beiden Fällen wird Nebenläufigkeit nicht benötigt um das gewünschte Programmverhalten zu erhalten. Vielmehr muss die Nebenläufigkeit durch geeignete Synchronisation so kontrolliert werden, dass nur "sinnvolles" Verhalten erlaubt ist. Tatsächlich liegen den meisten Konsistenzbedingungen für nebenläufige Programme, wie etwa der *Linearisierbarkeit* [HW90], sequenzielle Spezifikationen zugrunde.

Wir erinnern an das Beispiel aus Abbildung 1. Dieses Beispiel hat nur zwei mögliche Ausführungen, $x := x + 1; x := x + 2$ and $x := x + 2; x := x + 1$. Müssen wir wirklich beide Betrachten, und—im Wesentlichen—die möglichen Zwischenzustände $x = 2$ and $x = 1$ in den Beweisannotationen auflisten? Da die Addition *kommutativ* ist, spielt die Reihenfolge in diesem Beispiel keine Rolle. Wir können eine Reihenfolge wählen und argumentieren, dass die andere Reihenfolge "äquivalent" ist. Um Kommutativität im Allgemeinen auszunutzen um Beweise auf eine Untermenge von Verzahnungen zu reduzieren müssen folgende Fragen beantwortet werden: (1) Wie bestimmen wir die Kommutativität einzelner Operationen? (2) Wie spezifizieren wir die zu betrachtende Untermenge an Verzahnungen? (3) Wie begründen wir, dass alle anderen Verzahnungen implizit abgedeckt sind?

In einer grundlegenden Arbeit stellte Lipton [Li75] das Konzept von *rechts Movern* und *links Movern* vor. In Lipton's ursprünglicher Definition ist eine Operation ein rechts Mover, wenn die Operation in jeder Ausführung nach rechts (d.h. später) über jede Operation

eines anderen Threads permutiert werden kann, ohne den Wert einer Programmvariable im Endzustand zu verändern. Analog ist eine Operation ein links Mover, wenn sie über Operationen in anderen Threads nach links (d.h., früher) permutiert werden kann. Beispielsweise ist das Erwerben eines Locks ein rechts Mover, und das Freigeben eines Locks ein links Mover. *Lipton's Reduktion* ersetzt eine Sequenz $c_1;\ldots;c_n$ mit dem atomaren Kommando $[c_1;\ldots;c_n]$, falls für ein i alle $c_1,\ldots,c_{i-1}$ rechts Mover und alle $c_{i+1},\ldots,c_n$ links Mover sind (c_i is uneingeschränkt). In [FQ03] wurde die theoretische Idee der Mover in ein Typsystem zum Beweis von Atomarität von Methoden in einer nebenläufigen objektorientierten Sprache übernommen. Dieses Typsystem basiert auf der fixen Klassifizierung von bestimmten Operationen als Mover. Die Arbeit am QED Verifizierer [EQT09] stellte das Konzept der *bedingten atomaren Aktionen* vor, welche eine atomare Aktion nicht nur als Menge von Zustandsübergängen ausdrücken, sondern zusätzlich mit einer *Schranke* versehen, welche eine Bedingung angibt die zur Ausführung der Aktion gelten muss. Schranken erfassen kontextbezogene Information, wodurch wir Kommutativitätseigenschaften von atomaren Aktionen in isolation erheben können. Anstatt einer a priori Klassifizierung spezifischer Operationen als Mover, können wir *Movertypen* an bedingte atomare Aktionen durch paarweise Kommutativitätsanalyse zuweisen. Durch die Verwendung von Schranken identifizierten Elmas et al. [EQT09] die *Abstraktion* als symbiotisches Pendant zur Reduktion, wodurch iterative Programmvereinfachung ermöglicht wird. Abstraktion einer atomaren Aktion (d.h., Stärkung der Schranke oder Schwächung der Übergangsrelation) kann deren Movertyp stärken und dadurch Reduktion ermöglichen. Reduktion formt neue grobkörnige atomare Aktionen, welche wiederum abstrahiert werden können um Reduktion erneut anwendbar zu machen.

Reduktion erleichtert die Konstruktion von induktiven Invarianten, da Invarianten für ein reduziertes Programm um ein Vielfaches einfacher sein können als für das ursprüngliche Programm, was den Aufwand für die Reduktion mehr als kompensiert. In dieser Dissertation präsentieren wir neue reduktionsbasierte Beweisregeln für asynchrone Programme und verteilte Systeme. Dadurch ermöglichen wir Reduktionsbeweise für eine völlig neu Klasse von Programmen.

Verfeinerung. Programmentwicklung durch *schrittweise Verfeinerung* ist die Idee zur Entwicklung eines Programms durch sukzessive Verfeinerung einer abstrakten Spezifikation zu einer konkreten Implementierung. Alternative kann eine konkrete Implementierung sukzessive abstrahiert werden, um eine abstrakte Spezifikation zu beweisen. Oder der top-down und bottom-up Ansatz wird kombiniert. Formale Verifikation durch schrittweise Verfeinerung wird, in der Theorie, seit langem zur Konstruktion verifizierter nebenläufiger Programme vorgeschlagen (z.B. [BvW98, Sc97, Ro01]).

Zurück zum Modell der Transitionssysteme, sei K ein konkretes und A ein abstraktes Transitionssystem. Ein Standardansatz zum Beweis dass K eine korrekte Implementierung von A ist bedient sich einer *Verfeinerungsfunktion* von konkreten Zuständen von K zu abstrakten Zuständen von A. In einer sogenannten *Vorwärtssimulation* wird gezeigt, dass die Verfeinerungsfunktion jedem konkreten Schritt in K einen abstrakten Schritt in A zweist. Dadurch wird jedes konkrete Verhalten durch ein abstraktes Verhalten gerechtfertigt.

Die Arbeit in dieser Dissertation ist im Kontext des Verifikationssystems CIVL, erstmals Beschrieben von Hawblitzel et al. [Ha15]. Als reduktionsbasierter Verifizierer verfechtet CIVL die Verfeinerung von nebenläufigen Programmen über mehrere *Abstraktionsschichten* hinweg. Ein charakteristisches Designmerkmal von CIVL ist, dass alle Schichten in einem Verfeinerungsbeweis *strukturierte Programme* bleiben, d.h., Programme mit Prozeduren, imperativem Kontrollfluss, strukturellem Parallelismus, und asynchroner Nebenläufigkeit. Dies ist im Gegensatz zu bisherigen verfeinerungsbasierten Verifikationssystemen, wie TLA+ [La02b] oder Event-B [Ab96], welche nebenläufige Systeme als flache Transitionssysteme repräsentieren. Zwei Vorteile der Repräsentation als strukturierte Programme sind (1) der natürliche Brückenschluss von abstrakten Modellen zu realen Implementierungen, und (2) der Erhalt der im Programmtext enthaltenen Strukturinformation. In CIVL entspricht jeder Beweisschritt einer kleinen, vereinfachenden Prgrammtransformation, welche die atomaren Aktionen im Programm immer grobkörniger und abstrakter, und das Programm insgesamt immer weniger nebenläufig machen. Die benötigten Invarianten zur Begründung einzelner Beweisschritte sind vergleichsweise simpel, und die ermöglichte Dekomposition macht Beweise einfacher zu konstruieren und wiederzuverwenden. Die Implementierung von CIVL übersetzt das Verifikationsproblem in eine Menge an modularen Verifikationsbedingunen, welche von einem automatischen Theorembeweiser überprüft werden.

3 Beiträge der Dissertation

In dieser Dissertation beschäftigen wir uns mit formalen Methoden, die Entwickler und Programmierer dabei unterstützen, zuverlässigere nebenläufige Systeme zu entwickeln. Wir stellen neue Techniken, Methodiken, und Werkzeuge zur rigorosen Analyse und Verifikation solcher Systeme vor. Grob gesagt fallen die Beiträge dieser Dissertation in zwei Kategorien. Als erstes präsentieren Kapitel 2 und Kapitel 3 die formalen Grundlagen zur Entwicklung eines verfeinerungsbasierten Verifizierers, welcher alle Abstraktionsschichten als strukturierte Programme repräsentiert. Als zweites präsentieren Kapitel 4 and Kapitel 5 neue reduktionsbasierte Beweisregeln, welche neuartige simple Beweise für asynchrone Programme ermöglichen. Jedes Kapitel entspricht einem Forschungspapier, welches bei einer top-tier Konferenz veröffentlicht wurde.[3]

Kapitel 2: Layered Concurrent Programs [KQ18]
Bernhard Kragl, Shaz Qadeer
CAV 2018 (30th International Conference on Computer Aided Verification)
Konferenzrang: A (ERA) / A1 (Qualis)

Kapitel 3: Refinement for Structured Concurrent Programs [KQH20]
Bernhard Kragl, Shaz Qadeer, Thomas A. Henzinger
CAV 2020 (32nd International Conference on Computer Aided Verification)
Konferenzrang: A (ERA) / A1 (Qualis)

[3] Die Quelle für Konferenzreihungen ist `http://www.conferenceranks.com`. ERA reiht von A (=am besten) bis C (=am schlechtesten). Qualis reiht nach A1 (=am besten), A2, B1, ..., B5 (=am schlechtesten).

Kapitel 4: Synchronizing the Asynchronous [KQH18]
Bernhard Kragl, Shaz Qadeer, Thomas A. Henzinger
CONCUR 2018 (29th International Conference on Concurrency Theory)
Konferenzrang: A (ERA) / A2 (Qualis)

Kapitel 5: Inductive Sequentialization of Asynchronous Programs [Kr20b]
Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha O. Mutluergil, Shaz Qadeer
PLDI 2020 (41st ACM SIGPLAN Conference on Programming Language Design and Implementation)
Konferenzrang: A (ERA) / A1 (Qualis)

Zusammengefasst leistet diese Dissertation [Kr20a] folgende technische Beiträge.

Layered Concurrent Programs. Wir präsentieren *Layered Concurrent Programs* (dt. *geschichtete nebenläufige Programme*) (Kapitel 2), einen kompakten Formalismus zur Repräsentation aller Programme in einem mehrschichtigen Verfeinerungsbeweis als eine einzige syntaktische Einheit. Dadurch wird exzessive Duplikation von jenen Programmteilen vermieden, welche sich in einzelnen Beweisschritten nicht ändern.

Yield Invariants. Wir präsentieren *Yield Invariants* (Kapitel 3), ein neues Spezifikationsidiom, welches induktive Invarianten benennt, parametrisiert, und als wiederverwendbare Einheiten zusammenfasst. So können Yield Invariants spezifisch für eine bestimmte Aufrufstelle instanziiert werden (ähnlich wie Prozeduren). Yield Invariants kombinieren die Präzision von Invarianten à la Owicki-Gries und die Kompaktheit von Rely-Guarantee Spezifikationen [Jo83]. Die Portierung existierende Beispiele zur Benutzung von Yield Invariants ergab signifikant Verbesserungen von Beweiskomplexität und Performance.

Verfeinerung über strukturierte Programme. Wir präsentieren eine mächtige Beweisregel zur schrittweisen Verfeinerung, welche das Verifikationsproblem über strukturierte Programme in modulare Verifikationsbedingungen zerlegt (Kapitel 3). Diese Beweisregel integriert Yield Invariants mit einem flexiblen System für *linear Berechtigungen* zur Verbesserung der Beweislokalität. Wir unterstützen sowohl die *Einführung* als auch die *Elimination* von lokalen und globalen Variablen, sowie die modulare Abstraktion von rekursiven Prozeduren.

Synchronization. Wir präsentieren ein Reduktionsprinzip namens *Synchronization* (dt. *Synchronisation*) (Kapitel 4), welches die Umwandlung von asynchronen Prozeduraufrufen zu synchronen Prozeduraufrufen ermöglicht. Dadurch wird das Verifikationsproblem erheblich erleichtert, da wir nicht mehr über asynchrone Berechnungsschritte zu einem beliebigen späteren Zeitpunkt in einer Berechnung schließen müssen, sonder (für Beweiszwecke) so tun können, als ob die Berechnung sofort ausgeführt wird.

Inductive Sequentialization. Wir präsentieren ein Reduktionsprinzip namens *Inductive Sequentialization* (dt. *Induktive Sequenziellisierung*) (Kapitel 5), welches das Schließen über ein verteiltes System auf eine einzige sequenzielle Ausführung des Systems reduziert. Wir zeigen, dass selbst komplizierte Protokolle wie Paxos simple sequenzielle Reduktionen erlauben. Unsere Beweise mittels Inductive Sequentialization sind um ein Vielfaches

einfacher als existierende Beweise mittels standard induktiver Invarianten, da Inductive Sequentialization das Problem umgeht, über beliebig viele und beliebig lange verzahnte Ausführungen zu Schließen.

Pending Asyncs. Wir erweitern bedingte atomare Aktionen mit der Idee von *Pending Asyncs* (dt. *ausstehende asynchrone Aktionen*). Mittels Pending Asyncs spezifizieren atomare Aktionen nicht nur Zustandsänderungen globaler Variablen, sondern auch die Erzeugung asynchroner Berechnungen. Pending Asyncs wurden erstmals für die Arbeit an Synchronization präsentiert, und bildeten anschließend die technische Grundlage für Inductive Sequentialization. Die Beweisregel in Kapitel 3 beschreibt die "Erzeugung" von Pending Asyncs, wohingegen Kapitel 4 und Kapitel 5 Techniken zur "Eliminierung" von Pending Asyncs beschreiben.

CIVL Verifizierer. Alle Techniken in dieser Dissertation wurden in unserem Verifizierungssystem CIVL[4] implementiert, welches als Teil von Boogie[5] frei verfügbar ist. Konkret bildete die Theorie aus Kapitel 2 und Kapitel 3 die Basis für ein neues Design und Implementierung von CIVL, und die Techniken aus Kapitel 4 and Kapitel 5 sind als neue Beweistaktiken verfügbar, welche symbiotisch mit den bereits existierenden Taktiken integriert wurden. Mittels unserer Implementierung demonstrierten wir in zahlreichen Fallstudien die Anwendbarkeit und Nützlichkeit unserer Verifikationsmethodik.

4 Ausblick

In dieser Dissertation präsentierten wir einen neuen Ansatz zur deduktiven Verifikation nebenläufiger Programme. Die vorgestellte Verfeinerungsmethodik über strukturierte nebenläufige Programme ermöglicht die schrittweise Abstraktion von feinkörniger Prozeduren zu grobkörnigen atomaren Aktionen. Die Konstruktion von formalen Beweisen durch Benutzer wird in überschaubare Teilprobleme untergliedert, und die Beweisprüfung durch einen Computer wird in modulare Verifikationsbedingungen übersetzt. Wir sind überzeugt, dass formal verifizierte Implementierungen nur dann gängige Praxis werden können, wenn Programmierung und Verifikation in eine vereinte Aktivität zusammengeführt wird. Unsere Arbeit fördert diese Zusammenführung durch die einheitliche Repräsentation aller Abstraktionsschichten eines mehrschichtigen Verfeinerungsbeweises (von konkreter Implementierung zu abstrakter Spezifikation) im selben Formalismus, und die kompakte Repräsentation aller Abstraktionsschichten und deren Zusammenhang in einem einzigen geschichteten nebenläufigen Programm.

Wir integrierten neuartige reduktionsbasierte Programmvereinfachungen in unsere Methodik, welche asynchrone Programmausführungen *synchronisieren* bzw. *sequenziellisieren*, und dadurch die Intuition von Programmierern über simple Verzahnungen ausnutzen. Wir haben unser Verifikationsverfahren in zahlreichen anspruchsvollen Fallstudien angewendet und dabei demonstriert, dass unser Verfahren viel einfachere Beweise erlaubt als bisher bekannte Beweise. Besonders entscheidend ist, dass diese Vereinfachung nicht nur die

[4] https://civl-verifier.github.io
[5] https://github.com/boogie-org/boogie

Komplexität des finalen Beweises betrifft, sondern die intellektuelle Herausforderung der eigentlichen Beweiskonstruktion!

Insgesamt stellt diese Dissertation einen bedeutenden Fortschritt auf dem Stand der Technik der Programmierung zuverlässiger nebenläufigen und verteilten Systemen dar. Einige unabhängige Forscher publizierten bereits Artikel unter der Verwendung unseres Verifizierers CIVL. Obgleich die formale Verifikation von anspruchsvollen nebenläufigen Algorithmen und realistischen Implementierungen durchaus eine Herausforderung bleiben wird, so ermöglicht es unser Dekompositions- und Strukturierungsmechanismus, dass sich Programmierer dieser Herausforderung bestens gewappnet stellen können.

Literaturverzeichnis

[Ab96] Abrial, Jean-Raymond: The B-book - assigning programs to meanings. 1996.

[Ba05] Barnett, Michael; Chang, Bor-Yuh Evan; DeLine, Robert; Jacobs, Bart; Leino, K. Rustan M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: FMCO. 2005.

[BvW98] Back, Ralph-Johan; von Wright, Joakim: Refinement Calculus - A Systematic Introduction. Graduate Texts in Computer Science. 1998.

[CGR07] Chandra, Tushar Deepak; Griesemer, Robert; Redstone, Joshua: Paxos made live: an engineering perspective. In: PODC. 2007.

[EQT09] Elmas, Tayfun; Qadeer, Shaz; Tasiran, Serdar: A calculus of atomic actions. In: POPL. 2009.

[Fl67] Floyd, Robert W.: Assigning Meanings to Programs. Proceedings of Symposium on Applied Mathematics, 19, 1967.

[FQ03] Flanagan, Cormac; Qadeer, Shaz: A type and effect system for atomicity. In: PLDI. 2003.

[Ha15] Hawblitzel, Chris; Petrank, Erez; Qadeer, Shaz; Tasiran, Serdar: Automated and Modular Refinement Reasoning for Concurrent Programs. In: CAV. 2015.

[Ho69] Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. Commun. ACM, 12(10), 1969.

[HW90] Herlihy, Maurice; Wing, Jeannette M.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst., 12(3), 1990.

[Jo83] Jones, Cliff B.: Specification and Design of (Parallel) Programs. In: IFIP Congress. 1983.

[KQ18] Kragl, Bernhard; Qadeer, Shaz: Layered Concurrent Programs. In: CAV. 2018.

[KQH18] Kragl, Bernhard; Qadeer, Shaz; Henzinger, Thomas A.: Synchronizing the Asynchronous. In: CONCUR. 2018.

[KQH20] Kragl, Bernhard; Qadeer, Shaz; Henzinger, Thomas A.: Refinement for Structured Concurrent Programs. In: CAV. 2020.

[Kr20a] Kragl, Bernhard: Verifying Concurrent Programs: Refinement, Synchronization, Sequentialization. Dissertation, IST Austria, 2020.

[Kr20b] Kragl, Bernhard; Enea, Constantin; Henzinger, Thomas A.; Mutluergil, Suha Orhun; Qadeer, Shaz: Inductive sequentialization of asynchronous programs. In: PLDI. 2020.

[La98] Lamport, Leslie: The Part-Time Parliament. ACM Trans. Comput. Syst., 16(2), 1998.

[La02a] Lamport, Leslie: Paxos Made Simple, Fast, and Byzantine. In: OPODIS. 2002.

[La02b] Lamport, Leslie: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. 2002.

[Li75] Lipton, Richard J.: Reduction: A Method of Proving Properties of Parallel Programs. Commun. ACM, 18(12), 1975.

[MP90] Manna, Zohar; Pnueli, Amir: A Hierarchy of Temporal Properties. In: PODC. 1990.

[OG76] Owicki, Susan S.; Gries, David: Verifying Properties of Parallel Programs: An Axiomatic Approach. Commun. ACM, 19(5), 1976.

[OO14] Ongaro, Diego; Ousterhout, John K.: In Search of an Understandable Consensus Algorithm. In: USENIX Annual Technical Conference. USENIX Association, 2014.

[Ow75] Owicki, Susan S.: Axiomatic Proof Techniques for Parallel Programs. Dissertation, Cornell University, 1975.

[Ro01] de Roever, Willem P.; de Boer, Frank S.; Hannemann, Ulrich; Hooman, Jozef; Lakhnech, Yassine; Poel, Mannes; Zwiers, Job: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science. 2001.

[Sc97] Schneider, Fred B.: On Concurrent Programming. Graduate Texts in Computer Science. 1997.

[St01] Stoica, Ion; Morris, Robert Tappan; Karger, David R.; Kaashoek, M. Frans; Balakrishnan, Hari: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM. 2001.

[vRA15] van Renesse, Robbert; Altinbuken, Deniz: Paxos Made Moderately Complex. ACM Comput. Surv., 47(3), 2015.

[Za12] Zave, Pamela: Using lightweight modeling to understand Chord. Comput. Commun. Rev., 42(2), 2012.

Bernhard Kragl ist ein Applied Scientist in der S3 Automated Reasoning Group bei Amazon Web Services (AWS). Seine Forschungsinteressen sind Programmiersprachen und formale Methoden für die Entwicklung zuverlässiger Computersysteme. Er promovierte am IST Austria unter der Betreuung von Thomas A. Henzinger. Seine Dissertation beschäftigt sich mit Beweistechniken für nebenläufige und verteilte Systeme. Zuvor schloss er mit Arbeiten zum automatischen Schließen und Theorembeweisen ein Bachelor- und Masterstudium an der Technischen Universität Wien ab.